AFRL-OSR-VA-TR-2013-0208

CROSS-BOUNDARY SECURITY ANALYSIS

Thomas W. Reps
University of Wisconsin-Madison

April 2013
Final Report

AIR FORCE RESEARCH LABORATORY
AF OFFICE OF SCIENTIFIC RESEARCH (AFOSR)
ARLINGTON, VIRGINIA 22203
AIR FORCE MATERIEL COMMAND

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ORGANIZATION.

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
25-02-2013	FINAL REPORT	01-02-2009 to 30-11-2012

4. TITLE AND SUBTITLE
CROSS-BOUNDARY SECURITY ANALYSIS

5a. CONTRACT NUMBER

5b. GRANT NUMBER
FA9550-09-1-0279

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Thomas W. Reps

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
University of Wisconsin-Madison
Research & Sponsored Programs
12 North Park Street, Suite 6401

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
AFOSR
875 N Randolph St
Arlington, VA 22203

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)
AFRL-OSR-VA-TR-2013-0208

12. DISTRIBUTION/AVAILABILITY STATEMENT

DISTRIBUTION A: APPROVED FOR PUBLIC RELEASE

13. SUPPLEMENTARY NOTES

14. ABSTRACT

The goal of the project was to develop new methods to discover security vulnerabilities and security exploits. The research involved static analysis, dynamic analysis, and symbolic execution of software at both the source-code and machine-code levels. An aspect that distinguished the approach taken in the project from previous work was the attempt to uncover security problems due to differences in outlook between different levels of a system -- an approach called cross-boundary security analysis. The term refers both to (i) translation effects where the source-level outlook and the machine-code-level outlook differ, as well as (ii) differences in outlook between a source-level view of a component's API and the machine code that implements the component, which can sometimes allow a sequence of API calls to drive a program to a bad state. In both cases, one has two different artifacts that are supposed to have the same semantics, but whose semantics actually differ.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Thomas Reps
U	U	U	SAR		19b. TELEPHONE NUMBER *(Include area code)* 608-262-1204

Reset

Final Report for Grant FA9550-09-1-0279: Cross-Boundary Security Analysis

(01 Feb. 2009 – 30 Nov. 2012)

Thomas Reps
Computer Sciences Department
University of Wisconsin

Abstract

The goal of the project was to develop new methods to discover security vulnerabilities and security exploits. The research involved static analysis, dynamic analysis, and symbolic execution of software at both the source-code and machine-code levels.

An aspect that distinguished the approach taken in the project from previous work was the attempt to uncover security problems due to differences in outlook between different levels of a system—an approach called *cross-boundary security analysis*. The term refers both to (i) translation effects where the source-level outlook and the machine-code-level outlook differ, as well as (ii) differences in outlook between a source-level view of a component's API and the machine code that implements the component, which can sometimes allow a sequence of API calls to drive a program to a bad state. In both cases, one has two different artifacts that are supposed to have the same semantics, but whose semantics actually differ.

1 Objective and Technical Approach

Recent research in the fields of programming languages, software engineering, and program verification has led to new kinds of tools for analyzing programs for bugs and security vulnerabilities. In these tools, program analysis conservatively answers the question "Can the program reach a bad state?" Many impressive results have been achieved; however, the vast majority of existing tools analyze source code, whereas most programs are delivered as machine code. If analysts wish to vet such programs for bugs and security vulnerabilities, tools for analyzing machine code are needed.

The project "Cross-Boundary Security Analysis" focused on the analysis of machine code. The objective of the project was both to find security vulnerabilities (i.e., flaws in software), as well as inputs to the programs that capitalize on these flaws (exploits). The plan was also to do this for multiple hardware platforms (i.e., for multiple instruction sets).

The original insight behind the research undertaken in the project was that there are often differences in outlook when one examines different levels of a system. "Different levels" applies to several aspects of software: across module boundaries (e.g., a client application and the libraries that is uses), as well as across the source-code/machine-code translation boundary. However, the notion of "differences in outlook" was not well understood—and hence

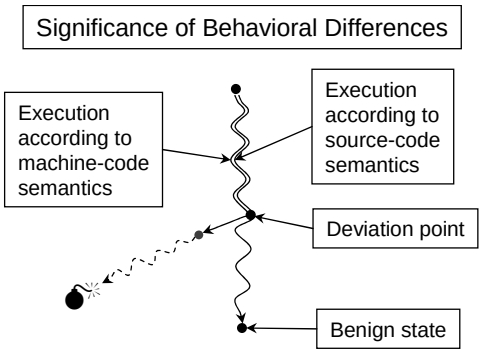

Figure 1: How to use a pair of models simultaneously to identify behavioral differences.

worthy of study. In each case, the plan was to analyze a pair of models simultaneously to identify behavioral differences—i.e,. semantic anomalies. There were multiple techniques to be applied, including static analysis, dynamic analysis, as well as a combination of symbolic evaluation and constraint solving.

Fig. 1 presents an idealization of the technical approach taken in the project. The first step is to find a deviation point—a place where the machine-code semantics and the source-level semantics differ—e.g., the source-level semantics is undefined, or underspecified, whereas the machine-code semantics follows that of the instruction-set semantics. Once a deviation point has been found, the goal switches to following the consequences in the machine-code to try to characterize the set of circumstances under which the flaw can be exploited.

2 Project Accomplishments

Work carried out during the project falls into five categories: §2.1 describes work on so-called "directed proof generation". §2.2 discusses techniques that were developed for machine-code analysis. §2.3 describes several approaches that were developed to make it easier, and more automatic, to develop program-analysis tools. §2.4 describes work carried out on analyzing concurrent programs. §2.5 discusses a few other results that were obtained during the course of the project.

2.1 Research on Directed Proof Generation

Directed proof generation is a program-analysis method that was developed in 2008 at Microsoft Research India for analyzing source-code programs [2]. Although it was originally developed for analyzing source code, directed proof generation has several characteristics that made it an attractive starting point for the project, the most important of which is that it performs a *goal-directed search* to see if a given target of interest can be reached.

The objective was to use directed proof generation on both source code and machine code simultaneously. The first hurdle was to make it work for machine code. The challenge faced was that the method for directed-proof-generation used in the Microsoft work uses many techniques that are unsound—i.e., lead to incorrect answers—if they were to be applied to machine code.

Consequently, the first two years of the project focused on the development of algorithms for directed proof generation that are suitable for use on machine code. A prototype tool,

called McVeto (for **M**achine **C**ode **VE**rification **TO**ol), was created, which incorporated the techniques developed (Conference publication (9)). What distinguishes the work on McVeto is that McVeto handles a large number of issues that have been ignored in previous work on program analysis, and would cause previous techniques to be unsound if applied to machine code.

Background on Directed Proof Generation. Given a program and a particular target location, directed proof generation returns either (i) an input for which execution leads to the target or (ii) a proof that the target is unreachable (or (iii) the algorithm does not terminate). It performs a goal-directed search to see if the given target of interest can be reached; if the target is unreachable, the outcome serves as a proof that the program is correct.

During the search, it maintains two approximations of the program's behavior, an under-approximation and an over-approximation. In essence, it plays both of them off each other: both are used at each stage to figure out which of them needs to be improved to create a better approximation. A combination of analysis techniques are used during this process: dynamic analysis (running tests), symbolic evaluation and constraint solving (to figure out whether the program can be driven down new paths), and (in the Wisconsin work) static analysis to make general improvements in the over-approximation.

The value of directed proof generation comes from the fact that it is capable of establishing the absence of vulnerabilities, as well as the definite presence of vulnerabilities. The two approximations are successively tightened until it is determined that either no input exists that can reach the target state, or such an input is found. If an input is found, it represents a definite vulnerability in the program. If the search terminates without finding such an input, it establishes that there is no vulnerability (at the target point).

Directed Proof Generation for Machine Code. In general, machine-code analysis is more difficult than source-code analysis, and directed proof generation is no exception. For instance, at the machine-code level, the analysis task is complicated by the fact that arithmetic and address-dereferencing operations are both pervasive and inextricably intermingled. On the other hand, machine-code analysis also offers new opportunities, in particular, the opportunity to track low-level, platform-specific details, such as memory-layout effects. Programmers are typically unaware of such details; however, they are often the source of exploitable security vulnerabilities.

Reps's group developed several innovations in methods for directed proof generation so that it could be applied to machine code. In particular, they developed a way to perform directed proof generation while avoiding a host of assumptions that are unsound in general, and that would be particularly inappropriate in the machine-code context, such as reliance on symbol-table, debugging, or type information, and preprocessing steps for (a) building a pre-computed, fixed, control-flow graph or (b) performing points-to/alias analysis. Instead, their method builds its abstraction of the program's state space on-the-fly. The initial over-approximation of the program has only two abstract states ("non-target" and "target"), and is gradually refined as more of the program is exercised.

The method does not require static knowledge of the split between code vs. data: it and uses a sound approach to disassembly. It performs disassembly during state-space exploration, but never on more than one instruction at a time. (That is, it does not have to be

prepared to disassemble collections of nested branches, loops, procedures, or the whole program all in one go, which is what leads conventional disassembly tools astray.) The method can even analyze programs with that use self-modifying code.

The method can also handle "instruction aliasing": programs written in instruction sets with varying-length instructions, such as x86, can have "hidden" instructions starting at positions that are out of registration with the instruction boundaries of a given reading of an instruction stream. The ability to deal with instruction aliasing can be important because it is sometimes used in exploiting security vulnerabilities.

Finally, they also developed a way to create such a tool for directed proof generation for machine code automatically, starting from a description of the instruction set to be supported. The net result is that a new version of McVeto for a different instruction set can be created easily: just write the specification of the instruction set's semantics; push a button; and receive your tool. (Given a specification, tool generation requires approximately 1 minute to generate some language-specific code, and approximately 1 hour to compile the resulting code.) Specifications of both the x86 and PowerPC instruction sets were implemented, and thus two versions of McVeto were created: McVeto/x86 and McVeto/PowerPC. (McVeto/x86 was the primary development testbed; McVeto/PowerPC was more of a proof-of-concept.)

Innovations in Directed Proof Generation for both Source Code and Machine Code. Some of the other innovations in directed proof generation that Reps's group developed during the project apply to both source code and machine code. One provides a way to avoid getting stuck in a trap in which directed proof generation keeps finding related spurious counter-examples (for the case i = 1, then the case i = 2, then . . .), which are then successively eliminated from the (approximate) representations of the program's state space that are maintained during directed proof generation. The technique that was worked out generalizes from *one* spurious counter-example so that a *whole collection* of related spurious counter-examples can be eliminated.

Another innovation involved the development of new symbolic techniques for the search that the verification tool carries out. The advantage of symbolic techniques is to be able to carry out operations "en masse", so that the verifier performs its work more efficiently.

Reps and his students also collaborated with Dr. Ken McMillan (Microsoft) to investigate how to extend McVeto to use the interpolant generator Foci to improve the search procedure used by McVeto. Unfortunately, Foci was not capable of handling the kinds of formulas that arise in directed test generation of machine code.

Supporting Directed Proof Generation for Source-Code Languages. A major goal was to support directed proof generation for source-code languages, in particular LLVM. Unfortunately, this work was only partially completed. LLVM compiles source code into an intermediate language with a far simpler semantics than most source languages, but still has some points where behavior is still undefined until compiled into lower-level code.

A semantic specification of LLVM "bit-code" was written in the TSL specification language. (Bit-code is one of the intermediate languages of LLVM; see §2.3 for a discussion of TSL and its capabilities.) The TSL compiler was used to generate an interpreter for LLVM from the semantic specification, which took care of the concrete-execution component needed for directed proof generation. Actually, the concrete interpreter was for a subset of LLVM

bit-code, and thus could run a subset of programs compiled to LLVM. (The issue was that LLVM uses data types that are not available in TSL, such as floating-point numbers.)

The intention was to use TSL to generate additional analysis components from the same TSL specification of LLVM—in particular, a symbolic-execution primitive to perform forward symbolic execution of LLVM bit-code programs. TSL had previously been used to develop symbolic-execution primitives for machine code (x86 and PowerPC), and the task that remained was to do the same for source code.

Unfortunately, to create such abstract interpreters for LLVM, they needed to extend TSL. Recursive data types in TSL are defined using discriminated unions; each discriminated-union type can consist of values constructed via multiple user-defined "operators". However, the current version of TSL only allows for abstract interpretation of discriminated unions that have a single operator. Because the semantic specification for LLVM has many recursive data types that involve discriminated unions over multiple operators, what remains to be done is to extend the TSL compiler to support enhanced methods for abstracting trees. They have mostly implemented such an extension, but it has not been tested thoroughly. An additional unfinished task is to create and implement useful abstractions of recursive data types, particularly ones that abstract the LLVM run-time stack.

2.2 Machine-Code Analysis

Several other projects that focused on machine-code analysis were also carried out:

- Technical report (3) describes a tool, called BCE (for "Botnet Command Extractor"), for automatically extracting botnet-command information from bot executables. The goal of the work on BCE was to address a major problem faced by malware vendors—namely, to analyze the behavior of bots, it is desirable to run the bot executables and observe their actions. However, to be able to execute bots, one needs proper input commands that trigger malicious behaviors. It is a difficult and time-consuming task to manually infer botnet commands from binaries.

  BCE uses a search strategy that improves on ones used in tools for directed test generation. Our experiments showed that the new search strategy developed for BCE yielded both substantially higher coverage of the parts of the program relevant to identifying bot commands, as well as lowered run-time.
- Conference publication (6) describes a tool, called PCCA (for "Producer-Consumer Conformance Analyzer"), that addresses the problem of identifying incompatibilities between two programs that operate in a producer/consumer relationship. PCCA attempts to (i) determine whether the consumer is prepared to accept all messages that the producer can emit, or (ii) find a counterexample: a message that the producer can emit and the consumer considers ill-formed. (PCCA was mainly supported under an ONR contract.)

A paper was also written to accompany an invited tutorial about machine-code analysis that was presented at the 2010 Int. Conf. on Computer-Aided Verification (CAV) (Invited paper (2)). Magazine article (1) also concerns machine-code analysis; it appeared in a forum whose readership is mainly practitioners.

2.3 Techniques to Make Program Analysis More Automatic

A substantial amount of work was carried out to develop techniques that make it easier to build program-analysis tools. With the techniques developed, it is possible to create tools more rapidly, with greater assurance that they are correct (because they are correct by construction), and that give more precise analysis results.

- Journal publication (1) is a lengthy journal paper that describes the TSL system. (TSL stands for "Transformer Specification Language"). TSL provides a systematic solution to the problem of creating retargetable tools for analyzing machine code. TSL is a tool generator—i.e., a meta-tool—that automatically creates different abstract interpreters for machine-code instruction sets. TSL provides help in automating the generation of the set of abstract transformers for a given abstract interpretation of a given instruction set. From a description of the concrete operational semantics of an instruction set, together with the datatypes and operations that define an abstract domain, TSL automatically creates the set of abstract transformers for the instructions of the instruction set. TSL advances the state of the art in program analysis because it provides two dimensions of parameterizability: (i) a given analysis component can be retargeted to different instruction sets; (ii) multiple analysis components can be created automatically from a single specification of the concrete operational semantics of the language to be analyzed.
- Journal publication (5) presents a novel technique developed in Reps's group to create implementations of the basic primitives used in symbolic program analysis: *forward symbolic evaluation*, *weakest liberal precondition*, and *symbolic composition*. The methods described in the paper have been implemented using TSL, which allowed correct-by-construction implementations of all three symbolic-analysis primitives to be obtained for both the x86 and PowerPC instruction sets.
- Nested-word automata [1] (NWAs) are a language formalism that helps bridge the gap between finite-state automata and pushdown automata. NWAs can express some context-free properties, such as parenthesis matching, yet retain all the desirable closure characteristics of finite-state automata.

  Conference publication (4) and Technical report (2) describe OpenNWA, a C++ library for working with NWAs, developed in Reps's group. The library provides the expected automata-theoretic operations, such as intersection, determinization, and complementation. It is packaged with WALi [5]—the Weighted Automaton Library (which also came out of Reps's group)—and inter-operates closely with the weighted-pushdown-system portions of WALi.
- Journal publication (6) introduced *view-augmented abstractions*, which specialize an underlying numeric domain to focus on a particular expression or set of expressions. A view-augmented abstraction adds a set of materialized views to the original domain. View augmentation can extend a domain so that it captures information unavailable in the original domain. The paper shows how to use finite differencing to maintain a materialized view in response to a transformation of the program state. Experiments showed that view augmentation can increase precision in useful ways.
- Pending submission (2) advances the state of the art in abstract interpretation of machine code. The method described in the paper tackles two of the biggest challenges

in machine-code analysis: (1) holding onto invariants about values in memory, and (2) identifying affine-inequality invariants while handling overflow in arithmetic operations over bit-vector data-types.

Symbolic Abstraction. Reps's group developed what they call "symbolic abstraction". This work has led to a collection of techniques that, in some cases, can attain the limit of precision that can be achieved using *any* algorithm that works with a given dataflow domain.

Symbolic abstraction (Conference publication (1), (2), (3), and (5), and Pending submission (1)) bridge the gap between (i) the use of logic for specifying program semantics and performing program analysis, and (ii) abstract interpretation. The connection hinges on the following question: given a formula φ in some logic L, how can one find the best value A in a given abstract domain $\mathcal{A}$ that over-approximates the meaning of φ (i.e., $[\![\varphi]\!] \subseteq \gamma(A)$)? An algorithm that, given φ, returns A is called a *symbolic-abstraction algorithm* (denoted by $A = \widehat{\alpha}(\varphi)$).

If you think of logic L as being a rich language for expressing meanings, and abstract domain $\mathcal{A}$ as corresponding to a logic fragment L' that is less expressive than L, then symbolic abstraction addresses a fundamental approximation problem: given $\varphi \in L$, find the strongest consequence of φ that is expressible in logic fragment L'. Recent work has shown that it is possible to give algorithms for finding strongest consequences for families of abstract domains that obey certain properties.

Symbolic abstraction is "dual use": in addition to providing improved techniques for program analysis, it also provides better methods for building decision procedures for computer-aided reasoning. There are really two aspects to this:

1. Recently, two SAT algorithms, called Conflict-Directed Clause Learning (CDCL) and Stålmarck's method, have been "reverse engineered" from the abstract-interpretation perspective ([3] and Conference publication (2)); this work shows that abstract interpretation has actually been used inside SAT algorithms without the designers of those algorithms being aware of that fact (or even of the concept of abstract interpretation). More precisely, what has been shown is that particular data types in the CDCL and Stålmarck algorithms are really instances of an abstract domain—in the sense of meeting a certain interface. The power of this observation is that one can then plug in new abstract domains in the same basic algorithm to obtain more powerful decision procedures ([4] and Conference publication (3)).
2. An algorithm for symbolic abstraction can be used for unsatisfiability checking (UNSAT): if $\widehat{\alpha}(\varphi) = \bot$ then φ is unsatisfiable. Similarly, an algorithm for symbolic abstraction can be used for validity checking: if $\widehat{\alpha}(\neg\varphi) = \bot$ then φ is valid.

2.4 Analysis of Concurrent Programs

Reps's group at Wisconsin did two pieces of collaborative work with Reps's colleague Prof. Shan Lu on analyzing concurrent programs. In both projects, a mixture of dynamic and static analysis was employed.

- Journal publication (2) focuses on concurrency bugs that result in program crashes, specifically buggy interleavings that directly cause memory bugs (NULL-pointer-dereferences, dangling-pointers, buffer-overflows, uninitialized-reads) on shared memory objects. (A study of the error-propagation process of real-world concurrency bugs showed that a common pattern accounted for 50% of non-deadlock concurrency bugs.)

The tool that was built, called ConMem, monitors program execution, analyzes memory accesses and synchronizations, and predictively detects these concurrency-memory bugs. ConMem was evaluated using seven open-source programs with ten real-world concurrency bugs. ConMem detected more tested bugs (9 out of 10 bugs) than a lock-set-based race detector and an unserializable interleaving detector, which detected 4 and 6 bugs, respectively. ConMem's false-positive rate was about one tenth of the other tools. ConMem has reasonable overhead suitable for use during development and testing.

- Conference publication (7) explored a consequence-oriented approach to improving the accuracy and coverage of state-space search and bug detection. The approach taken was to first statically identify potential failure sites in a program binary, and then use (backward) static slicing to identify critical read instructions that are highly likely to affect potential failure sites through control and data dependences. Finally, a single (correct) execution of a concurrent program is monitored to identify suspicious interleavings that could cause an incorrect state to arise at a critical read and then lead to a software failure.

  The tool that was built, called ConSeq, showed that the above approach produced several improvements in bug-detection coverage and accuracy. An evaluation on large, real-world C/C++ applications showed that ConSeq detects more bugs than traditional approaches and has a much lower false-positive rate.

Reps's group also completed several projects on model checking of concurrent programs:

- Journal publication (3) concerned automatically verifying safety properties of concurrent programs. In that work, the safety property of interest is to check for multi-location data races in concurrent Java programs, where a multi-location data race arises when a program is supposed to maintain an invariant over multiple data locations, but accesses/updates are not protected correctly by locks.

  A notable aspect of that work was to establish a new analysis principle, called *random isolation*, in which the analyzer uses a semantics that randomly choses a runtime object to be marked indelibly as a special object. The analyzer uses the "mark" to track that object separately from other objects (which are generally lumped together in a "summary object"). Because the marked object is chosen randomly, a proof that a safety property holds for it generalizes to *all* of the objects modeled by the accompanying summary object.
- Journal publication (4) presented a new decision procedure for verifying that a class of data races caused by inconsistent accesses on multiple fields of an object cannot occur (so-called *atomic-set serializability*). Atomic-set serializability generalizes the ordinary notion of a data race (i.e., inconsistent coordination of accesses on a *single* memory location) to a broader class of races that involve accesses on *multiple* memory locations.
- Journal publication (7) addressed the analysis of concurrent programs with shared memory. Such an analysis is undecidable in the presence of multiple procedures. One approach used in recent work obtains decidability by providing only a partial guarantee of correctness: the approach *bounds the number of context switches* allowed in the concurrent program, and aims to prove safety, or find bugs, under the given bound. That principle was improved in the work from Reps's group by means of a general

method to convert a concurrent program P, and a given context bound K, into a *sequential* program P_s^K such that the analysis of P_s^K can be used to prove properties about P.

The advantage of this approach is that it permits any *sequential* analyzer to be harnessed to analyze *concurrent* programs as well, under a context bound.

- Technical report (1) describes work that extended Journal publication (7) to permit an analyzer to exploit *induction* when analyzing a concurrent program.

2.5 Miscellaneous

Several other pieces of work related to program analysis and transformation were carried out with partial funding from AFOSR grant FA9550-09-1-0279.

Invited paper (1) describes DESKCHECK, a static analyzer that is able to establish properties of programs that manipulate dynamically allocated memory, arrays, and integers. DESKCHECK can verify *quantified invariants* over mixed abstract domains, e.g., heap and numeric domains.

Conference publication (8) addressed an obstacle to the use of recently developed decentralized information flow control (DIFC) operating systems. A DIFC operation system provides mechanisms for enforcing information-flow policies on the data used in a program. One obstacle to adoption of such systems is that, heretofore, DIFC operating systems provided only low-level mechanisms for application programmers to enforce desired policies—that is, the policy is created by sprinkling calls to operations in the enforcement API throughout an application. Because there is no first-class notion of a "policy" (separate from the code), it is difficult to even know what security policy is being enforced. The work described in Conference publication (8) introduced the idea of *policy weaving*: a policy is stated in a document separate from the program (and expressed in a suitable policy-specification language); a *policy-weaver* tool rewrites the program so that the specified policy will be enforced. (This piece of work served as the proof-of-concept and jumping-off point for a larger project at Wisconsin on policy weaving, conducted by Prof. Reps and his colleague Prof. Somesh Jha, which is funded under DARPA's CRASH program.)

Pending submission (3) defines a new variant of program slicing, called *specialization slicing*, and presents an algorithm for the specialization-slicing problem that creates an *optimal* output slice. An algorithm for specialization slicing is "polyvariant": for a given procedure p, the algorithm may create multiple specialized copies of p. In creating specialized procedures, the algorithm must decide for which patterns of formal parameters a given procedure should be specialized, and which program elements should be included in each specialized procedure.

3 Technology Transition

Prof. Reps is the co-founder, along with Prof. Emeritus Tim Teitelbaum of Cornell, of a company called GrammaTech, Inc. GrammaTech, which now has about 53 employees, is a leader in the creation of tools for software analysis and manipulation, with expertise in static and dynamic analysis; source code and machine code; and reverse engineering, information assurance, and information protection. GrammaTech has many defense contractors as customers, as well as research partnerships with others (e.g., Raytheon and Lockheed-Martin). In addition, GrammaTech has had multiple projects with NSA and several FFRDCs (San-

dia, SAIC, MIT Lincoln Labs, and IDA-CCS), as well as NRL. Thus, through GrammaTech, Reps has a pipeline for basic-research results obtained by his Wisconsin research group to be transitioned into the hands of working analysts in a relatively short time.

Over the past two decades, Reps has transferred through GrammaTech several pieces of technology to industrial use (including use by defense contractors), including (i) technology for analyzing compliance with coding standards, (ii) the CodeSurfer code-understanding tool (for C, C++, and x86), and (iii) the CodeSonar bug-finding tool for C and C++. Most recently, a major defense contractor licensed the x86 version of CodeSonar for use in supply-chain risk management. CodeSonar/x86 makes extensive use of the TSL tool that was developed by Reps's group.

Another example of technology transfer from Reps's group to the DoD via GrammaTech was highlighted in the 2007 report of the Defense Science Board Task Force on "Mission Impact of Foreign Influence on DoD Software" [6]. That report discussed the importance of "binary analysis" (analysis of machine code), and pointed both to GrammaTech's CodeSurfer/x86 tool, as well as to three specific analyses used in CodeSurfer/x86—affine-relation analysis (ARA), value-set analysis (VSA), and aggregate structure identification (ASI) [6, p. 63]. During 2001–2007, Reps and his then-student Gogul Balakrishnan developed ARA, VSA, and ASI for use on machine code, and collaborated with GrammaTech to build the CodeSurfer/x86 system. (From 2001–06, the work was mainly supported at Wisconsin under an OSD/ONR-sponsored MURI grant to Wisconsin under the Critical Infrastructure Protection—Software (CIP-SW) program. On the GrammaTech side, support came from an Air Force Rome Labs SBIR, as well as a variety of other sources.)

The results of the "Cross-Boundary Security Analysis" project on better techniques for analyzing machine code are being transferred through GrammaTech to customers of interest to the DoD in a similar fashion. In particular, GrammaTech is a heavy user of both TSL (Journal publication (1)) and WALi [5]. Thus, as planned, GrammaTech is serving as a pipeline to carry out the technology transfer needed to get such results into the hands of the DoD, and DoD contractors, in a relatively short time.

4 Archival Publications

All of the papers below, except two of the journal papers in press, are available from Prof. Reps's web site: http://pages.cs.wisc.edu/∼reps/. (The journal papers will be posted when they have appeared in print.)

4.1 Journal Publications

1. Lim, J. and Reps. T., TSL: A system for generating abstract interpreters and its application to machine-code analysis. To appear in *ACM Trans. on Program. Lang. and Syst.*
2. Zhang, W., Sun, C., Lim, J., Lu, S., and Reps, T., ConMem: Detecting crash-triggering concurrency bugs through an effect-oriented approach. To appear in *ACM Transactions on Software Engineering and Methodology 22*, 2 (2013).
3. Kidd, N., Reps, T., Dolby, J., and Vaziri, M., Finding concurrency-related bugs using random isolation. In *Int. Journal on Software Tools for Technology Transfer 13*, 6 (2011), 495–518.
4. Kidd, N., Lammich, P., Touilli, T., and Reps, T., A decision procedure for detect-

ing atomicity violations for communicating processes with locks. In *Int. Journal on Software Tools for Technology Transfer 13*, 1 (2011), 37–60.

5. Lim, J., Lal, A., and Reps, T., Symbolic analysis via semantic reinterpretation. In *Int. Journal on Software Tools for Technology Transfer 13*, 1 (2011), 61–87.
6. Elder, M., Gopan, D., and Reps, T., View-augmented abstractions. In *Electr. Notes Theor. Comput. Sci. 267(*1): 43–57 (2010).
7. Lal, A. and Reps, T., Reducing concurrent analysis under a context bound to sequential analysis. In *Formal Methods in System Design 35*, 1 (2009).

4.2 Invited Papers

1. McCloskey, B., Reps, T., and Sagiv, M.: Statically inferring complex heap, array, and numeric invariants. In Proc. Static Analysis Symposium (SAS), 2010.
2. Reps, T., Lim, J., Thakur, A.V., Balakrishnan, G., and Lal, A., There's plenty of room at the bottom: Analyzing and verifying machine code. In Proc. Computer-Aided Verification (CAV), 2010.

4.3 Conference Publications

1. Thakur, A., Elder, M., and Reps, T., Bilateral algorithms for symbolic abstraction. In Proc. Static Analysis Symposium (SAS), 2012.
2. Thakur, A. and Reps, T., A generalization of Staalmarck's method. In Proc. Static Analysis Symposium (SAS), 2012.
3. Thakur, A. and Reps, T., A method for symbolic computation of abstract operations. In Proc. Computer-Aided Verification (CAV), 2012.
4. Driscoll, E., Thakur, A., and Reps, T., OpenNWA: A nested-word-automaton library (tool paper). In Proc. Computer-Aided Verification (CAV), 2012.
5. Elder, M., Lim, J., Sharma, T., Andersen, T., and Reps, T., Abstract domains of affine relations. In Proc. Static Analysis Symposium (SAS), 2011.
6. Driscoll, E., Burton, A., and Reps, T., Checking compatibility of a producer and a consumer. In Proc. Found. of Software Engineering (FSE), 2011.
7. Zhang, W., Lim, J., Olichandran, R., Scherpelz, J., Jin, G., Lu, S., and Reps, T., ConSeq: Detecting concurrency bugs through sequential errors. In Proc. Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2011.
8. Harris, W., Jha, S., and Reps, T., DIFC programs by automatic instrumentation. In Proc. ACM Conf. on Computer and Communications Security (CCS), 2010.
9. Thakur, A.V., Lim, J., Lal, A., Burton, A., Driscoll, E., Elder, M., Andersen, T., and Reps, T.: Directed proof generation for machine code. In Proc. Computer-Aided Verification (CAV), 2010.

4.4 Magazine Articles

1. Anderson, P. and Reps, T., WYSINWYX: What You See Is Not What You eXecute. In Embedded Systems Design, Feb. 2010.

4.5 Technical Reports

1. Prabhu, P., Reps, T., Lal, A., and Kidd, N. Verifying concurrent programs via bounded context-switching and induction. TR-1701, Computer Sciences Department, University of Wisconsin, Madison, WI, November 2011.

2. Driscoll, E., Thakur, A., Burton, A., and Reps, T., WALi: Nested-word automata. TR-1675r, Computer Sciences Department, University of Wisconsin, Madison, WI, July 2010; revised Sept. 2011.
3. Lim, J. and Reps, T., BCE: Extracting botnet commands from bot executables. TR-1668, Computer Sciences Department, University of Wisconsin, Madison, WI, February 2010.

4.6 Pending Submissions

1. Thakur, A., Lal, A., Lim, J., and Reps, T., PostHat and all that: Attaining most-precise inductive invariants. Submitted for conference publication, Jan. 2013.
2. Sharma, T., Thakur, A., and Reps, T., An abstract domain for bit-vector inequalities. Submitted for conference publication, Jan. 2013.
3. Aung, M., Horwitz, S., Joiner, R., and Reps, T., Specialization slicing. TR-1776, Computer Sciences Department, University of Wisconsin, Madison, WI, October 2012. Submitted for journal publication.

5 Awards and Honors

1. 2011 ACM SIGSOFT Retrospective Impact Paper Award (for Reps, T., Horwitz, S., Sagiv, M., and Rosay, G., "Speeding up slicing" , 1994).
2. 2010 ACM SIGSOFT Retrospective Impact Paper Award (for Reps, T. and Teitelbaum, T., "The Synthesizer Generator" , 1984).
3. Kidd, N., Lammich, P., Touili, T., and Reps, T., A decision procedure for detecting atomicity violations for communicating processes with locks (SPIN Workshop, 2009) was invited for special submission to the Int. Journal on Software Tools for Technology Transfer.
4. Lim, J., Lal, A., and Reps, T., Symbolic analysis via semantic reinterpretation (SPIN Workshop, 2009) was invited for special submission to the Int. Journal on Software Tools for Technology Transfer.
5. Elder, M., Lim, J., Sharma, T., Andersen, T., and Reps, T., Abstract domains of affine relations (Static Analysis Symposium (SAS), 2011) was invited for special submission to ACM Trans. on Programming Languages and Systems.
6. As of February 2013, Thomas Reps was ranked 8th (citations) and 4th (field rating) on Microsoft Academic Search's list of most-highly-cited authors in the field of Programming Languages, and 23rd (citations) and 13th (field rating) on its list of most-highly-cited authors in the field of Software Engineering.

References

[1] R. Alur and P. Madhusudan. Adding nesting structure to words. In *Developments in Lang. Theory*, 2006.

[2] N.E. Beckman, A.V. Nori, S.K. Rajamani, and R.J. Simmons. Proofs from tests. In *Int. Symp. on Softw. Testing and Analysis*, 2008.

[3] V. D'Silva, L. Haller, and D. Kroening. Satisfiability solvers are static analysers. In *Static Analysis Symp.*, 2012.

[4] V. D'Silva, L. Haller, D. Kroening, and M. Tautschnig. Numeric bounds analysis with conflict-driven learning. In *Tools and Algs. for the Construct. and Anal. of Syst.*, 2012.

[5] N. Kidd, A. Lal, and T. Reps. WALi: The Weighted Automaton Library, 2007. www.cs.wisc.edu/wpis/wpds/download.php.

[6] Office of the Under Secretary of Defense for Acquisition, Technology, and Logistics. Report of the Defense Science Board Task Force on 'Mission Impact of Foreign Influence on DoD Software', September 2007. www.acq.osd.mil/dsb/reports/ADA486949.pdf.